Frame class in podio

(developing slides)

Thomas Madlener

February 17, 2022

`podio::Frame` - main purposes

- Act as a container that aggregates all relevant data
- Offer an easy to use and thread safe interface to access those data
    - (Immutable) read access to collections and *meta data*
    - Insert (via "destructive move") collections and *meta data*
    - Once inserted into the `Frame` it is immutable by design and no mutable access is granted afterwards
- Define an *interval of validity* or category (e.g. Event, Run, LumiSection) for the contained data

`podio::Frame` - why `Frame` and not `Event`

- The `Frame` is a more general concept
  - Functionality for reading, e.g. a `Run` is essentially the same as for reading an `Event`, the two differ mainly by their (data) content
- Having a concept of “lifetime” or “interval of validity” has some nice properties for dealing with *meta data*
- We can probably not exhaustively list experiment differences w.r.t. naming different things, but we can offer the necessary I/O functionality
  - Related to the *meta data* discussion, since we cannot foresee all the levels of metadata that are necessary
- Also works for experiments that have no clear notion of an “event” but rather deal with, e.g. read-out frames

Meta data - a definition

- *meta data* is all data that does not fit into either the EDM or the `podio::UserDataCollection`
  - Maybe **extra data** would be a better name, as it will probably be used to store additional data as wall as “true” meta data
  - Plus the distinction between *meta data* and *extra data* is a bit murky
- Usually implemented as some sort of generic key - value store
- Ideally there aren’t too many use cases for this in the end with podio based EDMs
  - Adding new datatypes is easy in podio
- Basis in podio is `podio::GenericParameters`
  - Offers a key - value storage for `int`, `float` and `std::string` as well as `vectors` thereof
  - (Plan to) not directly expose via the `Frame` interface

General functionality of a `Frame`

- Access to data read from file
- Possibility to add new data
- Ownership of the contained data
- Support for different I/O libraries
- Potential support for different *policies* with a single interface
    - Lazy unpacking (`prepareAfterRead` and potential decompression) of collections
    - How to handle missing collections
    - (Key / name) collision behavior on collection insertion
- Thread safe for “general use”
    - Inserting and reading from multiple threads will / should not lead to a race condition
    - Probably via mutexes + locking

`podio::Frame` - in memory interface

```cpp
struct Frame {
  /// Get a const reference to a stored collection by name
  template<typename CollT>
  const CollT& get(std::string name) const;

  /// Put a colleciton into the collection and get a const
  /// ref back. coll needs to be moved into the Frame so
  /// that it will be invalidated outside
  template<typename CollT>
  const CollT& put(CollT&& coll, std::string name);

  /// Get a const reference to the meta data stored under
  /// the given key
  template<typename T>
  const T& getMetaData(std::string key) const;

  /// Store a COPY of the value as meta data with the given
  /// key. For symmetry with the put method return a const
  /// ref to the newly stored value
  template<typename T>
  const T& putMetaData(T val, std::string key);
};
```

*Ommitting a few `const&` for the `std::string` arguments as well as some `enable_if` machinery to enforce the destructive move

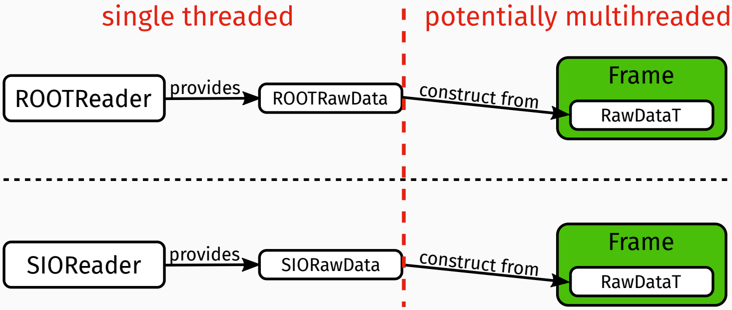
- The basic interface is rather simple
- The major points are
  - collections have to be ***moved* into the `Frame`** and become invalid after a call to `put`
  - It is not possible to get mutable data access
- Some additional functionality is needed for giving a `Writer` access to the stored data

*Method naming obviously up for discussion

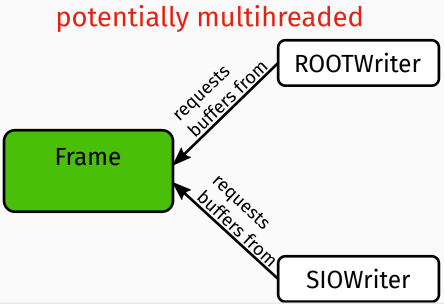
General I/O philosophy and assumptions

- I/O is assumed to be single threaded
  - Blocks until all requested data is written / read
- Readers proivde the data for a “complete” frame in (almost) arbitrary format
  - Can also be a subset of all collections
  - Combination of many frames (e.g. pile up mixing) into one not part of core podio
  - No “lazy reading”, i.e. once the data has left the reader, the frame is detached from it
- Writers request buffers to be written from the frame
  - Does not take ownership of these buffers
  - There can be multiple writers operating on one frame
- **podio provides the necessary building blocks for more complex workflows**
  - E.g. asynchronous reading / writing

I/O in diagrams



- Reading raw data and constructing a frame from it is a two step process
- Makes it possible to do unpacking on a separate thread



- Writing can happen with multiple threads, e.g. each writer on its own thread
- Writers can write different contents, e.g. SIM & RECO into separate files
    - Need one writer “per content”

podio::Frame - interface for I/O

```
struct Frame {
  /// Constructor taking some raw data provided by an
  /// external source
  template<typename RawDataT>
  Frame(std::unique_ptr<RawDataT> rawData);

  /// Get all CollectionBuffers to be written for the
  /// selected collections. Making sure all desired
  /// collections are put into the correct format before
  /// (i.e. prepared for write)
  std::vector<const podio::CollectionBuffers*>
  getBuffersForWrite(std::vector<std::string> names) const;

  /// Get the meta data container for writing
  const podio::GenericParameters& getMetaDataForWrite() const;
};
```

*Method naming up for discussion

- A `Frame` is constructible from (almost) arbitrary raw data
  - Can be different for each I/O system
  - Needs to provide access to the buffers of a desired collection
- A writer can request (a subset) of collection buffers to write
  - Frame takes care of preparing these buffers
  - Frame needs to be kept alive until writing all buffers is done $\rightarrow$ writer interface not concerned with this

The user perspective (single threaded)

```cpp
TrackCollection doTracking(const TrackerHitCollection& hits);
VertexCollection doVertexing(const TrackCollection& tracks);

podio::ROOTReader reader("hits.root");                          // Open a file to read from
podio::SIOWriter writer("reco_tracks.sio");                     // Open a file to write to

for (size_t i = 0; i < reader.getNEntries("event"); ++i) {      // At this low level we need to know the category
                                                                // that we want to read

  auto event = podio::Frame(reader.readNextEntry("event"));     // Create an event with the contents from the file

  const auto& hits = event.get<TrackerHitCollection>("hits");   // Get hits from event and
  auto tmpTracks = doTracking(hits);                            // do the tracking

  const auto& tracks = event.put(std::move(tmpTracks), "tracks");    // Store the tracks by moving them into the event
                                                                // Retain a const reference for later use
                                                                // tmpTracks now in "valid but undefined state"
                                                                // and is no longer usable

  const auto& vertices = event.put(doVertexing(tracks), "vertices"); // Temporaries don't need the explicit move

  auto recos = ReconstructedParticleCollection();
  event.put(std::move(recos), "recos");                         // Not keeping the const ref is also fine

  writer.writeEntry(event, "event");                            // Pass (a const ref to) the event to the writer
                                                                // Also the writer needs to know the category

}                                                               // frame goes out of scope and all data is destroyed
```

General functionality overview summary

- `Frame` acts as owning container of data and defines an /interval of validity/ (or category) for this data
    - Takes ownership of inserted data
    - Only gives immutable access to stored data
- Readers provide (almost) arbitrary raw data from which a `Frame` is constructed
    - The reader relinquishes ownership once the raw data leaves its control
    - No strict connection between a reader and a `Frame`
- The writers only get (references to) the buffers of data that should be written
    - Have to make sure that the write operation is done by the time the `Frame` is destroyed
- podio provides the main building blocks for constructing more complex workflows, but it will not offer “off-the-shelf” solutions for those

(Technical) Details

Collection meta data

- Collection meta data should be easily accessible from the collection directly
  - Currently have to go through the `EventStore`
- In LCIO each collection owns their own meta data container
  - Written separately for each event
- In the `Frame` approach all meta data is foreseen to be owned by a `Frame`
- **What is the correct lifetime for collection meta data?**
  - Do we want the distinction between “true” meta data and *extra* data with potentially vastly different lifetimes?
- Simplest solution is probably to somewhat follow the LCIO approach
  - Collection meta data lifetime == lifetime of the frame that containing collection
  - Pass requests from collection to frame and adapt keys under the hood
- Need to touch collection interface in any case

Frame policies

- We will not be able to satisfy everybody with one `Frame` implementation
- Offer a few (selected) customization points that allow podio users to alter some of the `Frame` runtime behavior
- Foresee that users want to define their own *policies*
- Possible customization points could be
  - The unpacking behavior - lazy (on demand) vs. eager at `Frame` construction
  - Collision handling on insert - throw an exception vs. overwrite existing vs ...
  - Handling of misssing data - throw an exception vs. default vs ...
  - Locking policy - e.g. have no locking at all if only used on a single thread
- Customization points should be as orthogonal to each other as possible

Frame implementation basics

TODO

RawDataT requirements

TODO

Supplementary Material

LCIO workflows / capabilities

- LCIO has the `LCEvent` that gives access to
  - The data stored in collections (defined by the EDM)
  - Some meta data (e.g. run & event number, weight, ...)
  - Meta/extra data in form of `LCParameters`
- `LCParameters` are essentially equivalent to podios `GenericParameters`
- Each collection has an instance of their own `LCParameters` attached

  → collection meta data has a lifetime of “event”
- Additionally there is the possibility to store collections of `LCGenericObjects`
  - Indexed based access to vectors of `int`, `float`, `double`
- LCIO also has an `LCRunHeader` with some meta data and `LCParameters`

Current podio vs. LCIO

- Overview table over the non-EDM possibilities of the two libraries

Use case	LCIO	podio
(arbitrary) user data	`LCGenericObject`	`UserDataCollection`
key-value @event	`LCParameters` of `LCEvent`	`GenericParameters` (event meta data)
key-value @collection	`LCParameters` of collections	`GenericParameters` (collection meta data)
key-value @run	`LCParameters` of `LCRunHeader`	`GenericParameters` (run meta data)

- In podio all the different levels of metadata are currently exposed via the `EventStore`
- In both cases the users get direct access to the whole `LCParameters` / `GenericParameters` object
    - Enforcing immutability would be extremely restricting for the users
    - Without immutability constraints → sequence point in multithreaded contexts

Some object sizes for overhead considerations

- Size of some internals of the `GenericParameters` as well as the probably largest `edm4hep` data type

object	size / bytes
`std::map<K, V>`	46
`std::unordered_map<K, V>`	56
`podio::GenericParameters`	46 * 4 or 56 * 4
`edm4hep::ReconstructedParticleData`	116
`edm4hep::ReconstructedParticleObj`	116 + 68
`edm4hep::ReconstructedParticleCollectionData`	280
`edm4hep::ReconstructedParticleCollection`	280 + 16

"Classic" polymorphism

"Library side"

- Defines an abstract interface

```
struct IReader {
  virtual std::string read() = 0;
};
```

"User side"

- Implementations have to inherit from `IReader`

```
struct ROOTReader : public IReader {
  std::string read() override { return "root"; }
};

struct SIOReader : public IReader {
  std::string read() override { return "sio"; }
};
```

- Usage requires pointer semantics

```
std::vector<std::unique_ptr<IReader>> readers;
readers.push_back(std::make_unique<ROOTReader>());
readers.push_back(std::make_unique<SIOReader>());

for (auto&& r : readers) std::cout << r->read();
```

Type erasure

"Library side"

- Essentially internalizes the abstract base interface

```cpp
class Reader {
  struct ReaderConcept {
    virtual std::string read() = 0;
  };

  template<typename R>
  struct ReaderModel final : public ReaderConcept {
    ReaderModel(R r) : m_reader(r) {}
    std::string read() final { return m_reader.doRead(); }
    R m_reader;
  };

  std::unique_ptr<ReaderConcept> m_self;
public:
  template<typename R>
  Reader(R r) :
    m_self(std::make_unique<ReaderModel<R>>(r)) {}

  std::string read() { return m_self->read(); }
};
```

"User side"

- Implementations are free standing classes that have to fullfill the interface required by the `ReaderModel`

```cpp
struct ROOTReader {
  std::string doRead() { return "root"; }
};

struct SIOReader {
  std::string doRead() { return "sio"; }
};
```

- Can be used with value semantics

```cpp
std::vector<Reader> readers;
readers.emplace_back(ROOTReader{});
readers.emplace_back(SIOReader{});

for (auto& r : readers) std::cout << r.read();
```